Федеральное государственное бюджетное образовательное учреждение
высшего образования
Санкт-Петербургский государственный университет

Потапова Вита Вячеславовна

МЕТАЭВРИСТИЧЕСКИЙ ПОИСК КАК СРЕДСТВО ВЫВОДА В ВЕРОЯТНОСТНОМ ПРОГРАММИРОВАНИИ

Выпускная квалификационная работа по направлению подготовки
035300 «Искусства и гуманитарные науки»

Профиль подготовки «Компьютерные науки и искусственный интеллект»

Научный руководитель: Потапов Алексей Сергеевич,
д. т. н., проф.

_____
подпись, дата

Санкт–Петербург
2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ....4

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ....6

1.1. Что такое метаэвристики....9

1.2. Описание методов метаэвристического поиска....11

1.2.1. Методы с одним состоянием....12

1.2.2. Популяционные методы....17

1.2.3 Некоторые другие виды методов....17

1.3. Вероятностное программирование....19

ВЫВОДЫ....22

ГЛАВА 2. РЕАЛИЗАЦИЯ ГЕНЕТИЧЕСКИХ ОПЕРАТОРОВ В ВЕРОЯТНОСТНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ....23

2.1. Вероятностные языки....23

2.1.1. Основные сведения о вероятностном программировании....23

2.1.2. Реализованный язык....26

2.2. Разработка генетических операторов для оптимизационных запросов....27

2.2.1. Мутации....27

2.2.2. Имитация отжига....29

2.2.3. Скрещивание....30

ВЫВОДЫ....34

ГЛАВА 3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ ОПТИМИЗАЦИОННЫХ ЗАПРОСОВ....35

3.1. Реализация вероятностного языка с оптимизационными запросами на c++....35

3.1.1. Реализация языка на основе конструкторов классов....35

3.1.2. Несоответствие поведения оптимизационных и семплирующих запросов....38

3.2. Автоматическое применение принципа минимальной длины описания....40

3.2.1. Байесовская бритва Оккама в вероятностном программировании. .40

3.2.2. Принцип минимальной длины описания....40

3.2.3. Автоматическая оценка сложности генеративных моделей....41

ВЫВОДЫ....42

ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНАЯ ПРОВЕРКА....43

4.1. Исследование эффективности реализации оптимизационных запросов....43

4.1.1. Полиномиальная аппроксимация.....43
4.1.2. Задача суммы подмножеств.....46
4.1.3. Предсказание целочисленных последовательностей.....48
4.2. Проверка применимости оптимизационных запросов в задачах с реальными данными.....49
ВЫВОДЫ.....53
ЗАКЛЮЧЕНИЕ.....54
СПИСОК ЛИТЕРАТУРЫ.....56

ВВЕДЕНИЕ

Настоящая работа посвящена оптимизационному подходу в области вероятностного программирования, которое является новой парадигмой машинного обучения. Объект нашего исследования – вероятностные языки программирования, а предмет – процессы вывода при интерпретации программ на вероятностных языках. Актуальность данной темы обусловлена недостатком эффективных средств вывода в вероятностных языках общего назначения, что ограничивает их применимость.

Цель работы – разработка интерпретатора вероятностного языка, поддерживающего оптимизационные запросы.

Гипотеза: оптимизационные запросы являются более эффективными, чем запросы на основе семплирования.

Задачи, которые необходимо решить, чтобы достигнуть поставленную цель, следующие:

- реализовать интерпретатор опорного языка, который будет расширяться вероятностной семантикой;
- реализовать метаэвристические методы оптимизации, включая имитацию отжига и генетическое программирование;
- реализовать поддержку оптимизационных запросов в интерпретаторе вероятностного языка на основе метаэвристических методов;
- экспериментально сравнить эффективность вероятностных языков на основе семплирования и оптимизационных запросов.

Основные источники представлены двумя ресурсами. Первый – книга «Probabilistic models of Cognition» [1], которая является описанием одного из общепринятых языков вероятностного программирования Church и его применением к задачам когнитивного моделирования. Второй – книга «The essentials of Metaheuristics» [2], где описываются основные концепции и методы метаэвристического поиска.

Основная часть работы состоит из 4 глав. В обзорной главе дан обзор проблемы поиска в области искусственного интеллекта. В заключении представлены выводы.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Мышление похоже на некоторую магическую способность, и его природа до сих пор остается необъяснимой. И подобно тому, как зритель наблюдает за фокусником, мы можем наблюдать за процессом мышления и со скептицизмом, и с любопытством. Если мы избираем вариант любопытства, то это похоже на то, как смотрит на эту тайну ученый. Перед ученым загадка мышления ставит сложный вопрос о том, как возникает мышление, и какие процессы им управляют.

Загадкой мышления занимается множество ученых из разных научных сфер – разгадать ее пытаются в когнитивной науке, в биологии и физиологии, в философии и в искусственном интеллекте. Ученые, занимающиеся построением искусственного интеллекта, как правило, верят, что попытки воссоздать процесс мышления могут быть достаточно успешными, чтобы увидеть, как же работает естественное мышление. Однако большинство из них давно не пытается воссоздать естественное мышление (Whole Brain Emulation не является частью ИИ), и цели все чаще сводятся к построению интеллектуальных машин, качество работы которых не будет уступать качеству действий человека в тех или иных ситуациях, а иногда даже будет несколько превосходить его.

Задача моделирования искусственного мышления оказалось довольно сложной, сложность была даже недооценена. Чего стоит только задача компьютерного зрения – одна из сложнейших сейчас, которую выдающийся специалист в области ИИ – Марвин Минский – поручил решить своему студенту за лето.

Первым доминирующим направление в области ИИ было направление, обычно обозначаемое термином «эвристическое программирование», в котором была принята метафора, что мышление – это поиск в пространстве решений.

Концепция поиска играла ключевую роль в психологической «лабиринтной» гипотезе мышления, которая была развита первыми

специалистами по ИИ, предмет исследования которых составляли интеллектуальные игры и доказательство теорем. Предполагалось, что суть интеллекта лежит в решении проблем, а сам процесс решения может быть представлен как поиск пути от исходных данных к ответу в пространстве возможных решений.

Существуют разные техники поиска, и самые простые из них – поиск в ширину и поиск в глубину – осуществляют полный перебор вариантов, что в случае большого пространства решений и огромных объемов данных становится вычислительно неэффективно.

Как оказалось, даже для сравнительно простых задач пространство решений чрезвычайно велико и исследование всего пространства простыми методами поиска невозможно. Очевидно, что наше мышление не ограничивается простым перебором вариантов. Для получения реально работающих программ необходимо использование неких правил или приемов, которые позволяют повысить эффективность программ, осуществляющих решение сложных задач. Такой способ или прием сокращения перебора – эвристика – отсекает неперспективные ветви на дереве вариантов.

Однако системы, построенные на основе идей эвристического программирования, оказываются неспособными самостоятельно решать любую новую задачу. Вряд ли подобные системы можно назвать действительно интеллектуальными. В связи с этим перед исследователями возникла проблема создания универсальных средств решения интеллектуальных задач.

Одна из программ, с помощью которой люди пытались моделировать процесс мышления, – Логик-теоретик (Logic-Theorist или LT) [3], показала, что способность находить доказательства логических суждений не требует больше возможностей, чем заложено в эту программу – читать, писать, сохранять, стирать и сравнивать. Однако эта программа смогла решать лишь ограниченный класс задач и не объяснила процесс мышления.

Одна из первых попыток создания такого универсального средства – «Общий решатель задач» (General Problem Solver, GPS), который был разработан на основе «Логика-теоретика». Хотя GPS тоже требует задания эвристик, они являются внешними по отношению к программе, в связи с чем применение GPS к новой предметной области не требует его перепрограммирования. На основе указанной информации GPS решает задачи, используя общий метод, называемый анализом целей и средств. Однако GPS по ряду причин не стал таким универсальным, как предполагали его создатели.

Проблемы поиска в непрерывных пространствах можно выделить в отдельный широкий класс проблем. Для этих проблем можно было бы использовать методы эвристического программирования, однако для того, чтобы эти методы оказались хоть сколько-нибудь эффективными, в них необходимо закладывать особые эвристики, опирающиеся на свойства непрерывных пространств, которые сильно влияют на организацию процессов поиска.

Задачи поиска в непрерывном пространстве зачастую сводятся к поиску экстремума некоторой целевой функции (поиск наилучшего пути также можно свести к этой задаче).

Дерево поиска при игре в го гораздо шире, чем в шахматах. И долгое время простыми эвристиками не удавалось достигнуть даже более-менее успешных результатов. Но существенный прорыв в игре в го был сделан с открытием методов Монте-Карло, имеющих стохастическую природу. Методы стохастического поиска также получили широкое применение и при решении задач поиска в дискретных пространствах.

Сейчас ведущие исследователи в области ИИ призывают вернуть данной области ее первоначальную цель – создание «мыслящих машин» [4, 5]. Подобласть ИИ, фокусирующуюся на этой цели, сейчас принято называть сильным или общим ИИ.

Два важнейших подхода в сильном ИИ – когнитивные архитектуры [6], в некотором смысле являющиеся очень продвинутыми вариантами GPS, и универсальный алгоритмический интеллект [7], в (вычислительно неэффективных) моделях которого устанавливаются ограничения, мешающие когнитивным архитектурам достичь универсальности. Эти подходы начинаются с очень разных отправных точек и иногда даже рассматриваются как несовместимые. Однако мы верим, что они должны быть объединены, чтобы строить ИИ, который будет как общий, так и эффективный. Однако нужен подход, который может помочь тесно объединить их на концептуальном уровне и уровне имплементации.

Вероятностное программирование могло бы стать подходящей основой для разработки такого подхода. В самом деле, с одной стороны, процедуры запросов в Тьюринг-полных вероятностных языках программирования могут быть использованы как прямые аппроксимации универсальной индукции и предсказания, которые являются центральными компонентами моделей универсального интеллекта. С другой стороны, вероятностное программирование уже успешно использовалось в когнитивном моделировании [1].

Для этого, однако, необходимо развить концепцию вероятностного программирования, включив в нее возможности по оптимизации целевой функции на основе метаэвристического поиска.

1.1. Что такое метаэвристики

Метаэвристики [2] – не самый удачный термин для обозначения методов: когда мы слышим «метадискуссия», мы думаем, что это дискуссия о дискуссиях. Таким же образом, когда мы слышим «метаэвристика», на ум приходит ассоциация «эвристики об (или для) эвристик», однако это не полно отражает суть этих методов. Вероятно, термин «black box optimization» подошел бы лучше, но здесь много лишней смысловой нагрузки. Поэтому часто используется термин «стохастическая оптимизация».

Алгоритмы стохастической оптимизации – общий класс методов, которые используют некоторую степень случайности при поиске максимально оптимальных решений сложных задач. Метаэвристические методы – самые общие из таких – применяются к очень широкому кругу задач.

Чем отличаются эвристики от метаэвристик? Эвристики позволяют находить решения для задач больших размеров, сокращая полный перебор, при этом они не гарантируют нахождения глобального результата. В то же время метаэвристики являются более мощными средствами оптимизации, так как имеют меньше вероятность застрять в локальном минимуме, могут применяться для решения практически любой оптимизационной задачи. Метаэвристики позволяют находить эвристики (параметры задачи) для конкретных задач (могут использовать в своей основе эвристики, которые применяются метаэвристиками ко всему пространству поиска). Но в некоторых случаях эвристики более быстрые, используются тогда, когда нужно найти хоть какой-то хороший вариант очень быстро. Бывает, когда эвристики не находят решения вовсе, но это случается довольно редко.

Метаэвристики относятся к методам «I know it when I see it». Эти алгоритмы используются, когда наши знания о задачах сильно ограничены – неизвестно заранее, как будет выглядеть оптимальное решение, нет определенного метода решения, слишком мало информации для использования эвристик и поиск «грубой силы» оказывается вне вопроса, так как пространство решений слишком велико. А если имеется кандидат на решение такой задачи – его можно проверить, и понять, насколько он оптимален. В таких случаях говорят, что решение известно, когда оно видно сразу.

Самое простое в таких случаях – случайный поиск: просто пробовать разные наборы поведения, пока есть время, и вернуть лучшее найденное решение. Но у такого варианта есть свои очевидные минусы, которые в некоторой степени решаются применением градиентного спуска.

Градиентный спуск использует идею о том, что похожие решения приводят к похожим по качеству результатам, и небольшие изменения будут приводить к маленьким улучшениям качества решения. Эта идея является центральной для всех метаэвристик: в самом деле, все они в той или иной мере используют градиентный спуск или случайный поиск.

Большинство классических методов оптимизации, в том числе и градиентный спуск, используют слишком сильные предположения о природе оптимизируемой функции. Метаэвристики делают гораздо более свободные предположения, или почти обходятся без них. Это значит, что метаэвристические методы очень общие, и к тому же, они используются, когда не работают другие известные методы. И так случается довольно часто в огромном, постоянно растущем числе сложных задач.

Существует множество разных видов метаэвристик: алгоритмы комбинаторной оптимизации (муравьиной колонии), параллельные (островная модель), многокритериальные и некоторые другие. Однако все они обладают выше описанными свойствами, по которым их и относят к метаэвристикам. Большое разнообразие таких методов позволяет для каждой конкретной задачи подбирать более подходящий метод, то есть более эффективный для конкретных наборов данных.

1.2. Описание методов метаэвристического поиска

В этом разделе большое внимание уделено основным метавристическим методам – методам с одним состоянием (градиентный спуск, имитация отжига, поиск с запретами). Существуют и другие классы методов (популяционные, методы комбинаторной оптимизации, параллельные, многокритериальные и некоторые другие), но в той или иной степени методы с одним состоянием являются основой методов других классов.

1.2.1. Методы с одним состоянием

Градиентный спуск

Градиентный спуск – простой прием, в некоторых модификациях называющийся метаэвристическим. Существует несколько вариантов этого метода. Не все из них определяются как метаэвристические, но в целом градиентный спуск лежит в основе всех метаэвристических алгоритмов. Суть его заключается в нахождении локального максимума (минимума) некоторой функции $f(\mathbf{x})$ путем движения по (против) направлению градиента функции $\nabla f(\mathbf{x})$ с некоторым небольшим шагом a.

Градиентный спуск в общем виде работает следующим образом. Сначала берется случайное решение. Затем производится маленькая случайная модификация предыдущего варианта и проверяется новая версия. Если модифицированный вариант оказывается лучше прежнего, старый вариант заменяется на новый. В противном случае, новый вариант забывается и старый вариант модифицируется снова. Так повторяется, пока не сработает критерий останова.

Одна из проблем градиентного спуска – это его время сходимости. Когда мы подберемся к максимуму функции, градиентный спуск просто перепрыгнет вершину и приземлится на другой стороне холма. Он может перепрыгивать вершину много раз в процессе приближения к максимуму.

Одна из причин этого явления заключается в том, что размер прыжков градиентного спуска целиком зависит от текущего наклона. Если наклон очень крутой, то и прыжок будет большим, даже если это будет бессмысленно. Один из способов решения этой проблемы – настройка градиентного спуска для рассматриваемой задачи путем изменения значения a (при этом также нередко делается нормировка вектора градиента на его модуль, что дает вектор перемещения $a\nabla f(\mathbf{x}) / |\nabla f(\mathbf{x})|$).

Возьмем очень малое a, и тогда градиентный спуск не перепрыгнет максимум, хотя будет очень долго взбираться по холму, пока не дойдет до вершины. При очень большом a градиентный спуск будет постоянно

перепрыгивать вершину, что тоже приведет к росту времени сходимости к максимуму, если он вообще будет найден. Поэтому нам нужно выбрать оптимальное значение a.

Мы также могли бы модифицировать алгоритм, изменяя другие факторы. Например, можно использовать метод Ньютона. Эта вариация на тему градиентного спуска включает дополнительно некоторый множитель. Такая модификация будет уменьшать a по мере приближения к точке с нулевым наклоном

Однако даже в этом случае сложно справиться с по-настоящему серьезной проблемой для подобных методов, состоящей в том, что градиентный спуск и метод Ньютона являются алгоритмами локальной оптимизации.

Для методов, которые мы рассмотрели, существует только один выход, как избежать локального оптимума. Можно изменить значение a на достаточно большое, чтобы алгоритм смог не только перепрыгнуть вершину текущего холма, но и приземлиться на другом холме. В случае с методом Ньютона можно запустить алгоритм по большому циклу, снова и снова пытаясь найти решение при различных начальных точках, и возвратить лучший найденный результат.

Имитация отжига

Одним из метаэвристических методов с одним состоянием также является имитация отжига, которая также относится к группе методов Монте-Карло. Алгоритм основывается на имитации физического процесса, который происходит при кристаллизации вещества, в том числе при отжиге металлов. Переход атома из одной ячейки в другую происходит с некоторой вероятностью, причём вероятность уменьшается с понижением температуры.

Текущее состояние оптимизируемой системы модифицируется по нормальному закону. Алгоритм может остаться в текущей точке, а может перейти в новое состояние. Вероятность перехода в новое состояние зависит

от температуры и разницы энергий, то есть значения оптимизируемой функции, в текущем и возможном новом состоянии и вычисляется в соответствии с некоторым распределением, например, с распределением Гиббса.

Классически применяется распределение Гиббса, в то время как применение распределения Больцмана дает методу новое название – «Больцмановский отжиг».

Алгоритм имитации отжига похож на градиентный спуск, но за счёт случайности выбора промежуточной точки должен будет попадать в локальные минимумы реже, чем градиентный спуск. При использовании логарифмического закона понижения температуры может быть гарантировано нахождение глобального минимума, с вероятностью, стремящейся к единице, однако на практике это требует слишком большого числа итераций, поэтому напрямую такой подход не используется.

Поиск с запретами (с ограничениями)

Поиск с запретами использует отличный от ранее рассмотренных методов подход к исследованию функции – он поддерживает историю недавно рассмотренных кандидатов на решение (которая называется «список табу») и не возвращается к этим решениям до того, как они будут достаточно далеко в прошлом. Этот метод работает только в дискретных пространствах. Но он достаточно просто обобщается на непрерывные пространства.

Сравнение методов

Проведем сравнение работы простых метаэвристических методов на простых функциях. На следующем графике (рис. 1), построенном для функции $(x - 2)^2 + y^2 + 198 \cdot \sin(x + y)$ видно, как работают данные методы.

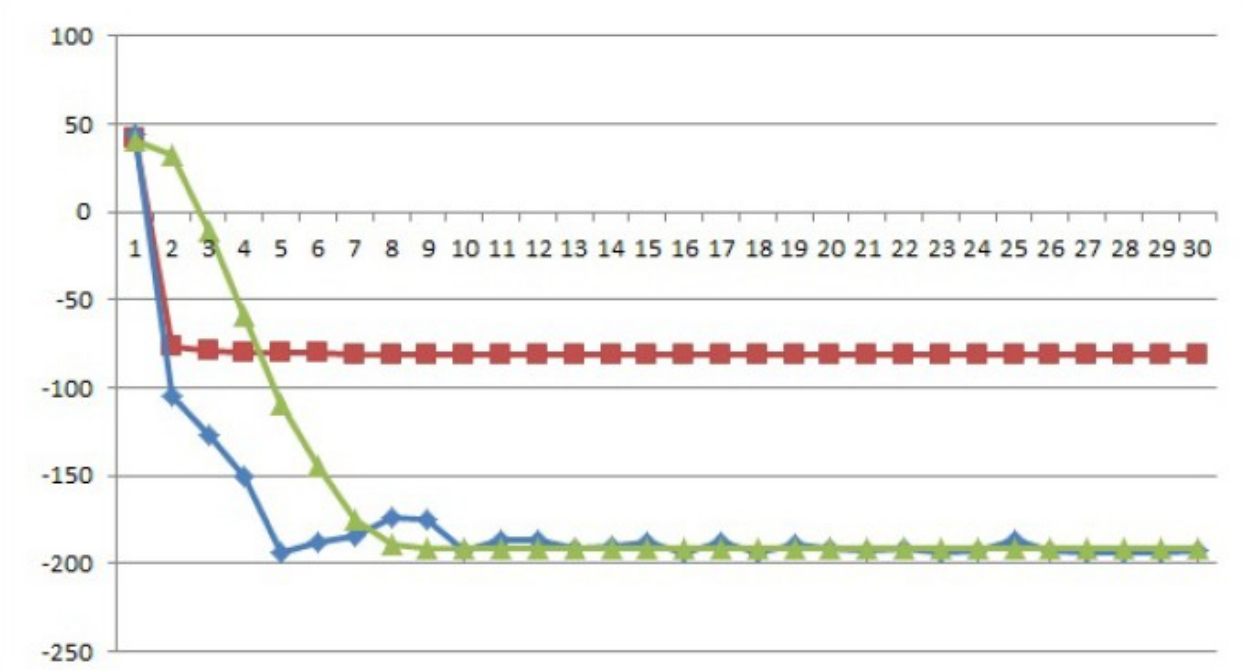

Рис. 1. Найденное минимальное значение градиентным спуском (красный), поиском с запретами (зеленым) и имитацией отжига (синим)

Быстрее всего своего минимума достигает градиентный спуск – ему потребовалось около 50 итераций, чтобы сойтись к значению $f(x, y) \sim -75$. Имитация отжига сошлась примерно в одну и ту же точку с поиском с ограничениями, равную приблизительно -200. Соответственно, имитация отжига и поиск с запретами показывают более эффективное поведение.

На данном графике видно типичное поведение имитации отжига – даже обнаружив некоторую хорошую точку, алгоритм может перейти в новую точку, качество которой хуже. Примечательно, что в конце концов вероятность перехода в менее оптимальное состояние сокращается, соответственно с течением времени происходит качественная оптимизация системы.

Поиск с ограничениями ведет себя стабильнее, чем имитация отжига, плавно спускаясь к минимуму функции. На первых итерациях может показаться, что градиентный спуск движется эффективнее по наклону функции, но преимущество поиска с ограничениями оказывается более очевидным, когда градиентный спуск застревает в локальном минимуме, а поиск с запретами спускается к глобальному экстремуму.

На следующем графике (рис. 2) оптимизировалась функция $(x - 2)^2 + y^2 + 5 + 1{,}5xy\sin(x + y)$. В этот раз алгоритмы начинали свою работу с разных точек. Градиентный спуск использует фиксированную начальную точку для расчета градиента, в то время как имитация отжига и поиск с запретами определяют точку, с которой будут начинать, случайным образом. Как и на предыдущем графике, красная линия – градиентный спуск, синяя линия показывает работу имитации отжига, а зеленая – поиска с ограничениями.

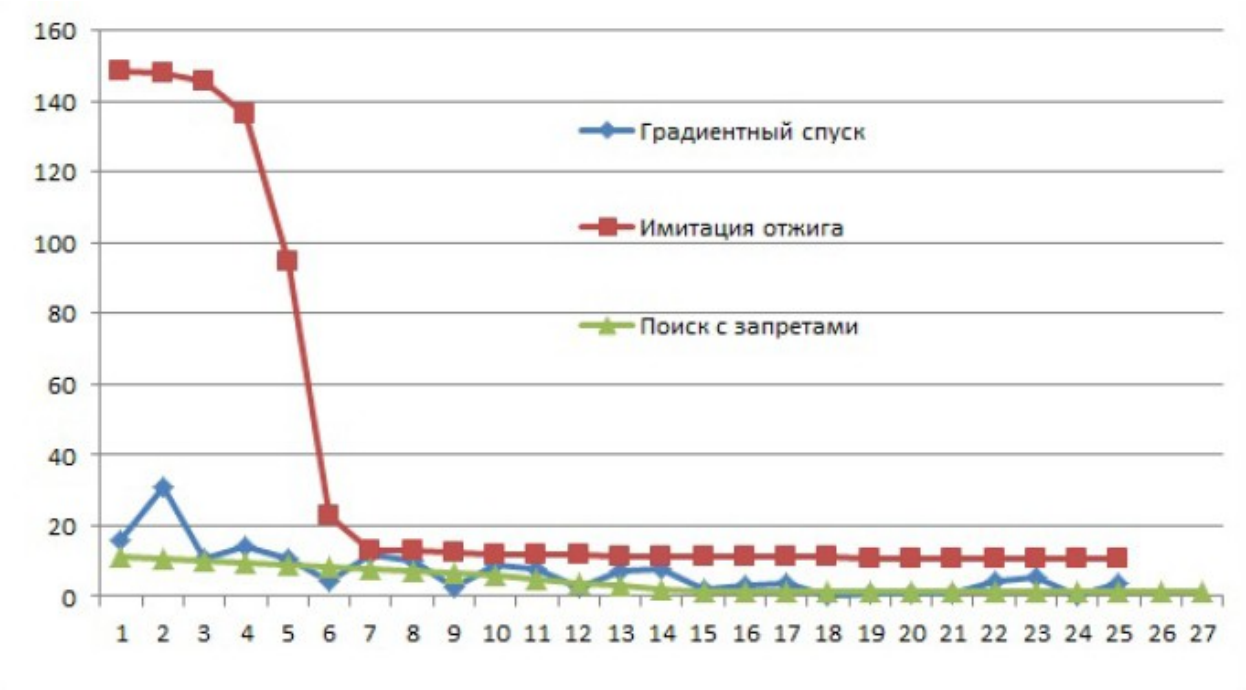

Рис. 2. Найденное минимальное значение градиентным спуском (красный), поиском с запретами (зеленым) и имитацией отжига (синим)

Как видно из графика, поведение градиентного спуска и поиска с запретами в целом стабильнее, чем имитации отжига. Однако, градиентный спуск снова менее качественно решает задачу оптимизации, чем его соперники, застревая в локальном экстремуме.

Стабильность поведения имитации отжига по большей части зависит от температуры, с которой начинается решение задачи. Чем меньше начальная температура, тем меньше прыжки в качестве решения, но больше риск остановиться на локальном экстремуме.

В целом, имитация отжига и поиск с ограничениями работают примерно одинаково эффективно и более качественно, сравнительно с простым методом градиентного спуска.

1.2.2. Популяционные методы

Эволюционные вычисления – это совокупность подходов к решению проблемы поиска (оптимизации), в основе которых лежат идеи, почерпнутые из эволюции в живой природе. Как хорошо известно, базовые элементы эволюции – наследственность, изменчивость и естественный отбор. Поскольку в процессе эволюции появляются все более приспособленные виды, эволюцию можно трактовать как поиск максимума некоторой фитнесс-функции.

Методы, основанные на использовании популяции, отличаются от методов, описанных в предыдущей части главы, тем, что работают с набором потенциальных решений, а не с единственным возможным решением. Каждое решение постепенно улучшается и оценивается. Основное отличие от простого параллельного метода локального поиска – потенциальное решение влияет на то, как будут улучшены другие решения.

Таким образом, основной механизм, отличающий популяционные методы, – это скрещивание.

1.2.3 Некоторые другие виды методов

Комбинаторная оптимизация

В задачах комбинаторной оптимизации решение представляет собой комбинацию уникальных компонент, выбираемых из, как правило, конечного и частого малого набора. Целью является поиск оптимальной комбинации компонент. Классическим примером задачи комбинаторной оптимизации является задача о рюкзаке или задача коммивояжера.

Для решения этих задач можно применять и рассмотренные ранее методы с одним состоянием, но большинство метаэвристик созданы для решения задач оптимизации в открытых пространствах поиска, а не при

наличии ограничений (в задаче коммивояжера и задаче о рюкзаке есть ограничения).

Примером алгоритмов комбинаторной оптимизации является широко известный алгоритм муравьиной колонии (Ant Colony Optimization), в котором компонентам присваиваются значения «исторически хороших» компонент. Так же есть вариант поиска с запретом, известный как управляемый локальный поиск (Guided Local Search), где штрафуются компоненты, использование которых ухудшает работу алгоритма. Эти методы подробно здесь не рассматриваются.

Параллельные методы

Использование метаэвристик может потребовать больших ресурсов. Например, при использовании генетических методов обычно используют не более 100 000 вычислений целевой функции за один запуск. Однако можно применять параллельные методы.

Ряд ученых считают, что применение параллельных методов само по себе оказывает благоприятное влияние на процесс оптимизации. Многие методы стохастической оптимизации могут быть распараллелены, хотя для некоторых это проще сделать, чем для других. Например, методы с одним состоянием распараллеливаются неестественным образом, в то время как популяционные методы более других готовы к такому способу работы.

Вот пять основных методов распараллеливания.

- Исполнять несколько запусков параллельно.

- Исполнять один запуск, в котором этап вычисления приспособленности (и возможно размножения и инициализации) обсчитывается в несколько потоков на одном компьютере

- Исполнять раздельные запуски, между которыми время от времени осуществляется обмен приспособленными особями. Островная модель.

- Исполнять один запуск, в котором на этапе вычисления приспособленности эта задача распределяется по удаленным компьютерам. Хозяин-раб или Клиент-сервер.

- Исполнять один запуск с процедурой селекции, предполагающей, что особи записаны в параллельный массив на векторном компьютере. Пространственно вложенные или мелкозернистые модели.

Эти пять методов можно комбинировать различным способом. Например, эффективно комбинируются островная модель и модель Хозяин-раб.

Многокритериальные методы

Часто возникают ситуации, когда необходимо оптимизировать не одну функцию качества или приспособленности, а сразу множество функций. Каждая из таких оптимизируемых функций называется критерием. Иногда можно найти решение, которое оптимально по всем критериям. Однако гораздо чаще возникает обратная ситуация, когда критерии не согласуются друг с другом. В таких случаях решением является компромисс по различным критериям.

Существует множество способов определить набор «наилучших вариантов», но наиболее эффективными являются алгоритмы, использующие множество Парето в пространстве возможных решений. Простейшим способом использовать это множество является Многокритериальные методы используют это множество, объединяя все критерии в одну функцию приспособленности, используя некоторое линейное соотношение.

Подробно рассматривать эти методы мы тоже не стали, так как они более эффективны при применении их к некоторым определенным задачам.

1.3. Вероятностное программирование

Вероятностное программирование – новое мощное средство в машинном обучении. Вероятностные языки программирования обеспечивают возможность определять генеративные модели декларативно без

необходимости вручную реализовывать модельно-зависимые алгоритмы вывода, заменяя их на встроенные общие методы вывода. В самом деле, вероятностные языки программирования очень удобны в использовании – достаточно просто задать генеративную модель в форме вероятностной программы и наложить нужные условия, над которыми будут вычислены апостериорные вероятности, и эти вероятности будут выведены автоматически. Это может использоваться не только в машинном обучении, но и при моделировании рассуждений на основе знаний [1]. При этом то, что раньше требовало разработки специальных машин вывода для экспертных систем, с помощью вероятностного программирования получается практически "бесплатно".

Многие решения в вероятностном программировании используют эффективные техники вывода для определенных типов генеративных моделей (например, байесовские сети) [8, 9]. Однако Тьюринг-полные языки более многообещающие в контексте сильного ИИ. Эти вероятностные языки допускают задание генеративные модели в форме произвольных программ, включая программы, которые генерируют другие программы на некотором предметно-зависимом языке. Таким образом, вывод над такими генеративными моделями автоматически выражается в выведение программ на языке, определенно пользователем. Поэтому та же самая машина вывода может быть использована для решения очень широкого спектра задач.

С одной стороны, качество работы общих методов вывода в вероятностных языках программирования может быть довольно низким даже для моделей с небольшим числом случайных выборов [10]. Эти методы обычно основаны на случайном семплировании (например методы Монте-Карло марковских цепей) [11, 12]. Существуют работы по разработке более сильных методов вывода в вероятностных Тьюринг-полных языках, но они эффективны не во всех случаях, например, для вывода программ, хотя некоторый прогресс в этом направлении был достигнут [13]. Поэтому необходимы более подходящие методы вывода, и генетическое

программирование (ГП) может быть рассмотрено как возможный кандидат, так как оно уже применялось для универсальной индукции [14] и когнитивных архитектур [15].

С другой стороны, широкая и простая применимость методов выводов в вероятностных языках также желаема для эволюционных вычислений. На самом деле, хотелось бы применять некоторые существующие реализации генетических алгоритмов просто определяя задачу вручную (которая может быть произвольной задачей комбинаторики или индуктивного программирования и т. п.) без разработки двоичных представлений решений или реализации проблемно-зависимых рекомбинаций и операторов мутаций (существуют некоторые попытки преодолеть это также в генетическом программировании, например, [16]).

Таким образом, интересно совместить общность вывода над декларативными моделями в Тьюринг-полных вероятностных языках и силу генетического программирования. Такая комбинация даст общее средство для быстрого прототипирования методов генетического программирования для произвольных предметно-зависимых языков путем простого задания функции генерирования программ на целевом языке. Это также может расширить инструментарий вероятностных языков программирования, так как традиционный вывод в вероятностном программировании выполняется для удовлетворения условиям, когда как цель генетического программирования – оптимизация фитнес-функции.

Мы представляем новый подход к выводу в вероятностных языках программирования, основанный на генетическом программировании (и в дополнении на имитации отжига), который применен к следам выполнения вероятностных программ. Каждый след выполнения программы является реализацией генеративной модели, заданной программой (результатом расширения программы в процессе оценивания). Рекомбинации и мутации следов выполнения программы гарантируют, что их результаты будут сгенерированы исходной вероятностной программой. Поэтому следы

выполнения программы используются как «универсальный генетический код» для произвольных генеративных моделей, и достаточно просто задать такую модель в форме вероятностной программы, чтобы выполнить эволюционные вычисления в пространстве его реализаций.

В статье [17] авторы установили, что «на существующие подходы к вероятностному программированию сильно повлиял байесовский подход к машинному обучению» и оптимизационный подход является многообещающим, так как «оптимизационные техники масштабируются лучше, чем техники поиска». То есть, данная работа также может быть рассмотрена как работа в этом направлении, которое намного меньше исследовано в области вероятностного программирования.

ВЫВОДЫ

Проблема поиска – одна из основных проблем области искусственного интеллекта. Проблема поиска фундаментальна и заключается в том, что даже для сравнительно простых задач пространство решений чрезвычайно велико и исследование всего пространства простыми методами поиска невозможно. Традиционное использование подхода на основе эвристического программирования подразумевает разработку специализированных методов под каждую задачу. Метаэвристические методы, в частности, имитация отжига и генетические алгоритмы, применимы к широкому классу задач и позволяют находить оптимальное решение, не застревая в локальных минимумах.

Вероятностное программирование – новое мощное средство разработки методов машинного обучения, однако в нем используются неэффективные методы вывода (поиска решений) на основе семплирования. Можно предположить, что метаэвристические методы поиска могут стать основой более эффективных процедур вывода в вероятностных языках программирования. Таким образом, цель работы можно поставить как разработку интерпретатора вероятностного языка, поддерживающего оптимизационные запросы.

ГЛАВА 2. РЕАЛИЗАЦИЯ ГЕНЕТИЧЕСКИХ ОПЕРАТОРОВ В ВЕРОЯТНОСТНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

2.1. Вероятностные языки

2.1.1. Основные сведения о вероятностном программировании

В традиционной семантике языков программирования исполняемая много раз программа со случайными выборами ведет к разным результатам. Главная идея, лежащая в основе вероятностного программирования, состоит в том, чтобы ассоциировать результат выполнения программы не с какими-либо определенными исходами, но с распределением вероятностей всех возможных исходов. Конечно, проблема состоит в том, чтобы представить и вычислить такие распределения для случайных программ со случайными выборами. Это может быть сделано напрямую только для некоторых не Тьюринг-полных языков, что не особо интересно в контексте проблематики общего ИИ.

Некоторые вероятностные языки программирования расширяют существующие языки, сохраняя их семантику как частный случай. Программы на этих языках обычно включают в себя вызовы (псевдо-) случайных функций. Вероятностные языки программирования используют расширенный набор случайных функций, соответствующих различным обычным распределениям, включая гауссово, бета, гамма, мультиномиальное и другие (например, в языке Church, который является расширением Scheme, реализованы такие функции как *flip*, *random-integer*, *gaussian*, *multinomial* и некоторые другие). Оценка таких программ со случайными выборами осуществляется тем же образом, как и оценка этой программы на базовом (не вероятностном) языке.

Однако программы на вероятностных языках расцениваются как генеративные модели, определяющие распределения над возможными возвращаемыми значениями [10], и их прямая оценка может быть

интерпретирована как взятие одного элемента выборки (испытания) из соответствующих распределений.

Важнейшее свойство вероятностных языков программирования – использование условных вероятностей, которое позволяет программисту накладывать условия (промежуточные или финальные) на результаты вычисления программ. Программы с такими условиями вычисляются с помощью условных (апостериорных) вероятностей, которые являются основой байесовского вывода.

Простейшая реализация условного вывода – режекторное семплирование, где результаты выполнения программы, не удовлетворяющие поставленному условию, отклоняются (не включаются в порожденное множество результатов, представляющих условное распределение). Условные вероятности, определенные вероятностными программами, соответствуют апостериорным вероятностям генеративной модели при имеющихся данных, поэтому их оценка очень полезна и может быть напрямую применена к задачам машинного обучения и вероятностному выводу. Такое режекторное семплирование может быть просто включено в большинство существующих языков программирования в качестве обычной процедуры, но такой вариант крайне неэффективен, так что применим только для очень малоразмерных моделей.

Можно использовать метод для эффективного вывода условных вероятностей без семплирования для ограниченного множества моделей, но общие методы вывода должны быть использованы для Тьюринг-полных языков. Один из таких широко используемых методов основан на Монте-Карло цепях Маркова, а именно, алгоритм Метрополиса-Гастингса. В частности, он используется в языке Church, в котором есть такие функции семплирования, как *rejected-query*, *mh-query* и некоторые другие.

Этот алгоритм использует стохастический локальный поиск для семплирования таких экземпляров, для которых данное условие будет оставаться верным. Для релиции этого метода для моделей, заданных

вероятностными программами, нужно включить небольшие изменения возвращаемых значений элементарных случайных процедур, вызываемых в этой программе, то есть эти значения должны быть мемоизированы [1].

Монте-Карло цепи Маркова могут быть более эффективны, чем режекторное семплирование, при вычислении апостериорных вероятностей. Однако без использования некоторых дополнительных техник, они могут быть так же эффективны, как и режекторное семплирование (или даже хуже из-за накладных расходов) в обнаружении первого подходящего результата семплирования. Можно легко проверить это на примере следующей простой программы на языке Church:

(mh-query 1 1 (define xs (repeat 20 flip)) xs (all xs)).

В этой программе определен список из 20 случайных булевых значений, и этот список возвращается, когда все значения равны true. Если заменить “mh-query 1 1” на “rejection-query”, время вычисления немного уменьшится. Однако обнаружение большого числа результатов семплирования методом mh-query будет значительно более быстрым, чем многократное выполнение rejection-query. Поэтому нерешенной здесь является задача нахождения первого допустимого экземпляра модели. Это делается вслепую как в Монте-Карло цепях Маркова, так и в режекторном семплировании.

В большом числе практических задач пользователь может преобразовать строгое условие на более мягкое или даже может изначально ставить задачу оптимизации некоторой функции. Поэтому процедуры запросов, которые принимают фитнес-функцию для оптимизации вместо строгого условия для удовлетворения, могут быть использованы как часть семплирования Монте-Карло цепей Маркова, также как и независимо использованы для решения оптимизационных задач.

2.1.2. Реализованный язык

Так как исследование пространства решений в вероятностном программировании требует, чтобы манипуляции со случайными выборами были сделаны во время интерпретации (оценивания) программы, разработка новых процедур запросов связана с внедрением в процесс интерпретации. Так как никакой язык программирования не поддерживает достаточно гибкого внешнего контроля этого процесса, для нас было проще реализовать новый интерпретатор. Однако мы решили не разрабатывать новый язык, но воспроизвести (используя язык Scheme как опорный) некоторый базовый функционал языка Church [1], включая некоторое число простых функций (+, -, *, /, and, or, not, list, car, cdr, cons и другие), некоторые случайные функции (flip, random_integer, gaussian, multinomial), декларацию переменных и функций (define, let), вызовы функций с рекурсией. Также “quote” и “eval” были реализованы.

```
'((define (tree)
    (if (flip 0.7) (random-integer 10)
                   (list (tree) (tree))))
  (tree))
```

Традиционные интерпретаторы языка Lisp возвращают разные результаты каждого запуска программы. Интерпретаторы вероятностных языков программирования содержат процедуры запросов, которые используются для вычисления апостериорных вероятностей или для семплирования, согласно специальному условию. Мы хотели расширить этот язык процедурами, основанными на генетическом программировании, которые принимают фитнес-функцию вместо строгого условия. Рассмотрим, как общие эволюционные операторы могут быть реализованы в этих условиях.

2.2. Разработка генетических операторов для оптимизационных запросов

2.2.1. Мутации

Чтобы скомбинировать генетическое программирование с вероятностными языками, мы считаем каждый запуск программы кандидатом на решение. Источник разнообразия кандидатов на решение возникает из различных исходов случайных выборов, сделанных в ходе интерпретации программы. Мутации состоят в небольших модификациях случайных выборов, произведенных в предыдущие запуски программы, которые остаются частью алгоритма Метрополиса-Гастингса. Все эти выборы должны быть запомнены и связаны с контекстом выполнения, в котором они производятся. Чтобы достичь этого, мы реализовали следующее представление следов выполнения программ, идея которого (но не реализация) схожа с той, что используется в реализации mh-query на языке Church [1].

В этом представлении каждое выражение из оригинальной программы во время рекурсивного оценивания преобразуется в структуру (struct IR (rnd? val expr) #:transparent), где IR – имя структуры, rnd? является #t когда случайный выбор делается во время оценивания выражения expr; val – результат оценивания (один результат семплирования из распределения, заданный expr). Программа – список выражений. Функция interpret-IR-prog была реализована для оценивания программ, заданных в символьной форме. Рассмотрим несколько примеров.

- (interpret-IR-prog '(10)) → (list (IR #f 10 10)) означает, что результат оценивания программы содержит только выражение 10, интерпретируемое само в себя, и этот результат не является случайным.
- (interpret-IR-prog '((gaussian 0 1))) → (list (IR #t -0.27 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))) означает, что результат оценивания (gaussian 0 1) был -0.27 и является случайным, в то время как его аргументы 0 и 1 были оценены в неслучайные значения 0 и 1.

- (interpret-IR-prog '((if #t 0 1))) → (list (IR #f 0 (list 'if (IR #f #t #t) (IR #f 0 0) 1))) означает, что только одна ветвь была оценена, в то время как другая сохранилась в оригинальной форме.
- В более сложном случае случайная ветвь может быть расширена в зависимости от результата оценивания стохастического условия: (interpret-IR-prog '((if (flip) 0 1))) → (list (IR #t 1 (list 'if (IR #t #f '(flip)) 0 (IR #f 1 1)))). Здесь (flip) вернул #f, так что вторая ветвь была расширена. Отметим также, что полное выражение является случайным, в противоположности предыдущему примеру.
- В определениях переменных только их значения преобразуются в IR: (interpret-IR-prog '((define x (flip)))) → (list (list 'define 'x (IR #t #f '(flip)))). Оценка определений выражается в изменениях среды как обычно. Let-выражения ведут себя похожим образом, но у них есть тело для оценки. Определения функций сохраняются неизменными (конечно, их оценка также выражается в изменениях среды).
- Символы, которые могут быть найдены в среде, заменяются на их значения: (interpret-IR-prog '((define x (random-integer 10)) x)) → (list (list 'define 'x (IR #t 5 (list 'random-integer (IR #f 10 10)))) (IR #t 5 'x)).
- Приложение не библиотечной функции заменяется своим телом и let-связкой своих аргументов:

(interpret-IR-prog '((define (f x) (random-integer x)) (f 3))) →

(list '(define (f x) (random-integer x))

(IR #t 1 (list 'let (list (list 'x (IR #f 3 3)))

(IR #t 1 (list 'random-integer (IR #f 3 'x))))))

Оцененная программа может быть рассмотрена как расширенная исходная программа, но с добавочной информацией обо всех случайных выборах, сделанных в процессе оценки. Эта программа может быть снова оценена, и ранее сделанные случайные выборы будут приняты во внимание во время этой переоценки. Мы расширили interpret-IR-prog таким образом, что она может принимать как исходные программы, так и их IR расширения (так как процесс переоценивания может происходить в еще не расширенных

ветвей, такая унификация необходима, чтобы иметь дело также со смешанными случаями).

Во время каждого следующего переоценивания расширенной программы детерминистические выражения не оцениваются снова, но их предыдущие значения также используются. Все стохастические выражения оцениваются таким же образом, как и во время первого запуска, кроме вызовов базовых случайных функций, поведение которых меняется. Эти функции модифицированы, чтобы принимать во внимание ранее возвращенные значения. Параметр скорости мутации p добавлен в interpret-IR-prog, чтобы представлять, насколько близкими должны быть новые значения к предыдущим. Например, предыдущий результат (flip) меняется с вероятностью, равной p. Переоценка (IR #t v (gaussian x0 s)), где v – ранее возвращенное значение, x0 – среднее и s – сигма, будет соответствовать (gaussian v (* p s)), но в других имплементациях может быть смещена к x0.

Например, результатом переоценки IR выражения (list (IR #t -0.27 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))), используя p=0.01, может быть (list (IR #t -0.26 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))).

2.2.2. Имитация отжига

Описанный интерпретатор уже достаточен, чтобы реализовать оптимизационный запрос, основанный на имитации отжига. Пусть дана программа, последнее выражение которой возвращает значение (фитнес-) функции энергии для минимизации. Тогда можно многократно оценить эту программу, отдавая предпочтение результатам, имеющим наименьшее значение последнего выражения.

Имитация отжига использует только один след выполнения программы (в форме IR выражения). Она выполняет interpret-IR-prog для генерации новых кандидатов на решение (с переходными вероятностями, полученными интерпретатором для данной программы и параметризованными температурой) и принимает их с вероятностью (/ 1 (+ 1 (exp (/ dE t)))), где dE

– разница энергий для решения-кандидата и текущего решения, и t – текущая температура (могут быть использованы другие вероятности принятия).

На каждой итерации решения-кандидаты генерируются до принятия (хотя число попыток ограничено), и температура уменьшается от итерации к итерации. Мы реализовали annealing-query на основе этого подхода.

2.2.3. Скрещивание

Скрещивание в форме кроссовера также использует следы выполнения программ. Однако ему нужна двойная переоценка двух расширений программы. Эти расширения интерпретируются совместно как одна программа, пока их структуры соотносятся (и они должны соотноситься, кроме вариаций, вызванных случайными выборами). Главная особенность – при повторном оценивании случайных функций необходимо принять во внимание ранее возвращенные значения от обоих родителей.

Например, в нашей реализации парный flip возвращает одно из предыдущих значений, и парная гауссиана возвращает (+ (* v1 e) (* v2 (- 1 e))), где v1 и v2 – предыдущие значения, и e – случайное значение на [0, 1] (можно сместить результат этого базового элемента кроссовера к начальному гауссовому распределению). Мутации введены одновременно с кроссовером в целях повышения эффективности.

Однако может быть встречена во время переоценки такая ветвь, которая не была еще расширена в одном или обоих родителях. В последнем случае такая ветвь просто оценивается как при первом выполнении. Иначе она переоценивается по тому родителю, для которого уже была расширена (без кроссовера, но с мутациями). Так как она не расширена по другому родителю, это расширение будет случайным и не оцениваться фитнес-функцией, поэтому она будет просто зашумлять информацию, полученную от другого родителя.

Интересно, что потомки могут содержать ранее расширенные, но ныне неиспользуемые ветви, которые могут быть активированы снова в

последующих поколениях в связи с единичными мутациями или даже кроссовером. Эти части расширенной программы напоминают мусорные ДНК, а их активация – быстрые генетические адаптации.

Может показаться, что разработанный подход построен для простой формы генетического программирования, применимой только для параметрических программ, но это не так, в виду того что в функциональных вероятностных языках можно просто запрограммировать произвольные генеративные модели, включая те, которые генерируют другие программы. Рассмотрим несколько простых примеров.

- Рассмотрим два результата оценки '((gaussian 0 1)) и их кроссовер:

```
(list (IR #t 0.113 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))
(list (IR #t 0.447 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))
(list (IR #t 0.236 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))
```

Можно увидеть, что результат кроссовера находится где-то между результатами обоих родителей.

- Теперь рассмотрим программу с одним условием: (if (flip) -10 (random-integer 10)). Во время кроссовера могут возникнуть разные ситуации. Например,

```
(list (IR #t 5
  (list 'if (IR #t #f '(flip)) -10
        (IR #t 5 (list 'random-integer (IR #f 10 10))))))
(list (IR #t 8
  (list 'if (IR #t #f '(flip)) -10
        (IR #t 8 (list 'random-integer (IR #f 10 10))))))
(list (IR #t 6
  (list 'if (IR #t #f '(flip)) -10
        (IR #t 6 (list 'random-integer (IR #f 10 10))))))
```

Здесь (flip) вернул #f для обоих родителей, так что парное переоценивание пошло по той же ветви, и оба случайных целочисленных результата были рекомбинированы.

```
(list (IR #t 9
  (list 'if (IR #t #f '(flip)) -10
          (IR #t 9 (list 'random-integer (IR #f 10 10))))))
(list (IR #t -10
  (list 'if (IR #t #t '(flip)) (IR #f -10 -10)
        '(random-integer 10))))
(list (IR #t 9
  (list 'if (IR #t #f '(flip)) (IR #f -10 -10)
          (IR #t 9 (list 'random-integer (IR #f 10 10))))))
```

Здесь разные ветви были оценены в следах выполнения родителей, и результат (flip) был случайно взят от первого родителя во время кроссовера. Так как случайное целое число не было еще оценено во втором родителе, результаты не были рекомбинированы, а напрямую взяты от первого родителя.

- Наиболее интересный случай кроссовера – рекурсивные стохастические процедуры. Рассмотрим следующую программу:

```
'((define (tree) (if (flip 0.7) (random-integer 10)
                                (list (tree) (tree))))
  (tree))
```

Расширение этой программы может продуцировать большие следы выполнения, поэтому рассмотрим результаты кроссовера на уровне значений последнего выражения.

'(6 9) + '(8 0) → '(7 6)

'(8 (6 2)) + 3 → '(8 (6 2))

'(7 9) + '((0 7) (7 4)) → '(7 (7 4))

'((3 (7 (1 7))) 5) + '((5 2) 2) → '((4 (7 (1 7))) 2)

Можно увидеть, что пока структура деревьев соотносится, два следа выполнения программ переоцениваются вместе и результаты random-integer сливаются в листьях, но когда структура расходится, поддерево случайно берется от одного из родителей (в зависимости от результата переоценки (flip 0.7)).

Этот тип кроссовера для сгенерированных деревьев автоматически следует из реализованного кроссовера для следов выполнения программ. Конечно, кому-нибудь может захотеться использовать другой оператор кроссовера, основанный на предметно-зависимых знаниях, например, чтобы обменивать произвольные поддеревья в родителях. Последнее сложно сделать в нашем представлении следов выполнения программ (и нужны дополнительные исследования, чтобы разработать более гибкое представление). С другой стороны, рекомбинация следов выполнения программ во время переоценивания гарантирует, что ее результаты могут быть порождены исходной программой, и также обеспечивает некоторую гибкость.

- Рассмотрим две различные реализации функции, порождающей список цифр (во втором случае должен быть вызван (digits '())).

```
(define (digits) (if (flip 0.2) '()
                     (cons (random-integer 10) (digits))))
(define (digits ds) (if (flip 0.2) ds
                        (digits (cons (random-integer 10) ds))))
```

Если размеры решений для рекомбинаций равны, конечный результат может быть одинаков для обеих программ, например, '(7 1 6 3) + '(3 0 3 2) → '(6 0 4 3). Однако рекомбинация следов выполнения программ со значениями '(8 1) и '(5 3 0 3 2) может привести к '(6 2 0 3 2) для первой программы и к '(5 3 0 6 1) для второй программы. С одной стороны, эта разница в результатах добавляет гибкость и некоторый контроль над кроссовером. С другой стороны, нужно понимать, как работает рекомбинация следов выполнения программ, чтобы строить модели в подходящей форме. Кажется, что у этой

проблемы нет универсального решения без анализа фитнес-функции, так как семантика модели спрятана в ней. Тем не менее, можно создать полезные решения на базе некоторой фиксированной схемы рекомбинации над следами выполнения программ.

На основе описанных операторов мутации и кроссовера был реализован evolution-query.

ВЫВОДЫ

Впервые были реализованы оптимизационные запросы для вероятностного языка программирования, основанные на метаэвристических методах – генетических алгоритмах и имитации отжига. Для этих целей в интерпретатор разработанного языка, который в ходе оценивания программ создает их следы выполнения, были добавлены процедуры мутации и скрещивания следов выполнения программ. Эти процедуры учитывают случайные выборы, сделанные в предыдущие запуски программы, но модифицируют их в соответствии с правилами работы генетических операторов.

Совокупность следов выполнения программы рассматривается как популяция решений-кандидатов, а результат их оценивания – как фитнесс-функция, в соответствии с которой происходит отбор решений. Имитация отжига была реализована на основе оператора мутации с убывающей от итерации к итерации скоростью мутации, что эквивалентно уменьшению температуры в Больцмановском отжиге.

Таким образом, были реализованы оптимизационные запросы – annealing-query и rejection-query.

ГЛАВА 3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ ОПТИМИЗАЦИОННЫХ ЗАПРОСОВ

В предыдущей главе была рассмотрена реализация оптимизационных запросов в вероятностном программировании на языке Scheme, но для практического использования в задачах компьютерного зрения и машинного обучения требуются более эффективные языки, одним из которых является C++, для которого, в частности, реализована открытая библиотека компьютерного зрения OpenCV. Поэтому, прежде чем переходить к практической проверке и более глубокому исследованию оптимизационного подхода к вероятностному программированию, опишем выполненную нами реализацию вероятностного языка на C++.

3.1. Реализация вероятностного языка с оптимизационными запросами на c++

3.1.1. Реализация языка на основе конструкторов классов

Мы ставим целью осуществить практическую, но общую реализацию вероятностного программирования, так что мы рассматриваем Тьюринг-полные языки и оптимизационную модель. Мы реализовали подмножество языка Scheme внутри C++, используя конструкторы классов вместо применения функций. Например, были реализованы с соответствующими конструкторами такие классы как *Define*, *Lambda*, *List*, *Cons*, *Car*, *Cdr*, *Nullp*, *ListRef*, *If* и другие. Все эти классы наследуются из класса *Expression,* у которого есть поле *std*::*vector<Expression *> children*, и таким образом можно построить дерево. Чтобы создать значения из выражений, был создан класс *Value* (с синонимом *V*). Этот класс используется для всех значений, динамически разрешая поддерживаемые типы.

Также были добавлены такие классы как *Add*, *Sub*, *Mult*, *Div*, *Gt*, *Gte*, *Ls*, *Lse* и прочие, и были перегружены такие операции как +, –, *, /, >, >=, <, <= и т.д., чтобы вызывать соответствующие конструкторы. Таким образом, можно написать, например,

Define(*f*, *Lambda*(*xs*, *If*(*Nullp*(*xs*), *V*(0), *Car*(*xs*) + *f*(*Cdr*(*xs*))))),

что соответствует

(define f (lambda (xs) (+ (if (null? xs) 0 (+ (car xs) (f (cdr xs)))))))).

Чтобы использовать символы *f* и *xs*, нужно объявить их как экземпляры класса *Symbol* (с синонимом *S*) или написать *S*(“*xs*”) вместо *xs*. Оператор скобок также перегружен, так что можно писать *f*(*xs*) вместо *Apply*(*f*, *xs*), где *Apply* – потомок класса *Expression*. Таким же образом можно написать *xs*[*n*] вместо *ListRef*(*xs*, *n*).

Были также добавлены классы, соответствующие базовым случайным распределениям, включая *Flip*, *Gaussian*, *RndInt* и некоторые другие.

В нашей библиотеке были также обернуты некоторые функции и структуры данных OpenCV. Была добавлена поддержка *cv*::*Mat* как базового типа, так что есть возможность написать, например, *Define*(*S*(“image”), *V*(*cv*::*imread*(“test.jpg”))). Все базовые перегруженные операции с *cv*::*Mat* наследуются, так что значения, соответствующие *cv*::*Mat*, могут быть сложены или перемножены с другими значениями.

Для того чтобы избежать огромных следов выполнения программ в процессе заполнения случайными значениями пикселей изображения (каждое такое значение становится узлом в следе выполнения программы), были включены такие классы как *MatGaussian* и *MatRndInt* для генерирования случайных матриц с целостными значениями. Такие случайные матрицы также могут быть порождены как отклонения от данных.

Упомянутые конструкторы различных классов используются просто для того, чтобы создавать выражения и выстраивать их деревья. Было также реализовано оценивание таких выражений. Дерево выражений в процессе оценки расширяется в след выполнения программы. Такой след выполнения программы тоже является деревом выражений, но со значениями, присвоенными его узлам. Процесс оценки и следы выполнения программ, реализованные в нашей библиотеке, являются такими же, как и реализованные в предыдущей главе на языке Scheme. Мы также заново

реализовали оптимизационные запросы, основанные на имитации отжига и генетическом программировании над следами выполнения программ. Например, можно написать следующую программу с результатом оценки, показанном на рис. 3

Symbol imr, *imb*;

AnnealingQuery(*List*()

<< *Define*(*imr*, *MatRndInt*(*img.rows*, *img.cols*, CV_8UC3, 256, *img*))

<< *Define*(*imb*, *GaussianBlur*(*imr*, *V*(11.), *V*(3.)))

<< *imr*

<< (*MatDiff2*(*imb*, *V*(*img*)) + *MatDiff2*(*imb*, *imr*) * 0.3));

Здесь сначала загружается некоторый объект *img* класса *cv*::*Mat*, запись *List*() << *x* << *y* << *z* … эквивалентна (list x y z …). Оператор << может быть использован, чтобы добавить элементы в список на этапе построения дерева выражений (не оценки). Переменная *i m r* создана как случайное трехканальное изображение с начальным значением в качестве *img*. *MatDiff2* вычисляет попиксельное СКО между двумя матрицами. *AnnealingQuery* – оптимизационный запрос, основанный на имитации отжига, который минимизирует значение своего последнего дочернего узла, и его возвращаемое значение устанавливается в соответствующее значение предпоследнего дочернего узла. Здесь второе слагаемое в оптимизационной функции предотвращает получение слишком зашумленного результата. Также был реализован *GPQuery*, основанный на генетическом программировании.

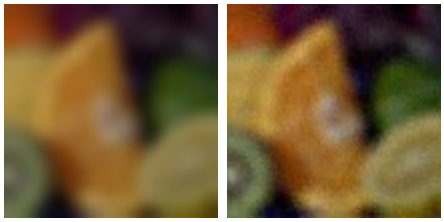

Рис. 3. Оригинальная размытая картинка и результат вывода

Имитация отжига не очень подходит для того, чтобы проводить поиск в пространстве изображений, но разумные результаты достигаются здесь за

несколько секунд. Также можно видеть, что общий код на C++ может быть легко использован совместно с нашей библиотекой вероятностного программирования. Конечно, код выполняется до или во время построения дерева выражений или после его оценки, но не в процессе оценки. Последнее может быть сделано путем расширения библиотеки новыми относительно простыми классами, но в рамках данной работы этого не потребовалось.

Деревья выражений могут быть использованы не только как фиксированные программы, написанные вручную, но и как строящиеся автоматически динамические структуры данных. Таким образом, такая библиотека может легко стать частью большей системы (например, когнитивной архитектуры).

3.1.2. Несоответствие поведения оптимизационных и семплирующих запросов

Оптимизационный подход эффективен для многих задач, и оптимизационные запросы даже без штрафования сложности могут быть применены в вероятностном программировании. Однако даже очень простые генеративные модели могут быть неэффективны в данном подходе. Рассмотрим следующую программу

```
Symbol xobs, centers, sigmas, n, xgen;
AnnealingQuery(List()
<< Define(xobs, V(4.))
<< Define(centers, List(3, -7., 2., 10.))
<< Define(sigmas, List(3, 1., 1., 1.))
<< Define(n, RndInt(Length(centers)))
<< Define(xgen, Gaussian(ListRef(centers, n), ListRef(sigmas, n)))
<< n
<< (xobs – xgen) * (xobs – xgen));
```

Казалось бы, эта программа должна просто возвращать число, соответствующее центру, ближайшему к *xobs*, в то время как *AnnealingQuery*

будет минимизировать расстояние от сгенерированного значения до центра класса. Однако оценка этой программы приводит к почти случайным индексам центров. Такая же модель хорошо работает на языке Church. Следующий запрос вернет распределение с $p(n=1)\approx 1$; и в случае (define centers '(-7., -2., 10.)) вернет $p(n=1)\approx p(n=2)\approx 0.5$.

```
(define (noisy-equal? x y)
(flip (exp (* -1 (– x y) (– x y)))))
(mh-query 100 100
    (define xobs 4)
    (define centers '(-7., 2., 10.))
    (define sigmas '(1., 1., 1.))
    (define n (random-integer (length centers)))
    (define xgen (gaussian (list-ref centers n) (list-ref sigmas n)))
    n
    (noisy-equal? xobs xgen))
```

Стоит отметить, что noisy-equal? должен применить *flip* для корректной оценки схожести, например, если нужно получить корректные апостериорные вероятности для *xgen*. В частности, должны быть заданы такие параметры, как дисперсия и точность. То есть, эти программы на языках C++ и Church на самом деле используют одинаковую информацию.

Неадекватный результат *AnnealingQuery* исходит из его возможности минимизировать данный критерий, уточняя значения всех случайных переменных, включая как *n*, так и *xgen* в этой модели. Намного проще подбирать *xgen* напрямую, так как его вероятность не учитывается в самом критерии. Эта проблема здесь может быть достаточно просто решена, если мы используем *AnnealingQuery* для минимизации расстояния от *n*-того центра до *xobs*. Программа будет проще, и результат будет правильный. Однако общая проблема останется. Она обнаружится в виде переобучения, невозможности выбрать подходящее число кластеров или сегментов в задаче кластеризации или сегментации, необходимости вручную определять *ad hoc*

критерии, и так далее. Как раз эти проблемы решаются использованием принципа МДО.

3.2. Автоматическое применение принципа минимальной длины описания

3.2.1. Байесовская бритва Оккама в вероятностном программировании

Такие вероятностные языки программирования, как Church, естественным образом поддерживают байесовскую бритву Оккама. Рассмотрим следующий очень простой пример.

```
(mh-query 1000 100
    (define n (+ (random-integer 10) 1))
    (define xs (repeat n (lambda () (random-integer 10))))
    n
    (= (sum xs) 12))
```

Здесь мы хотим, чтобы сумма неизвестного количества n случайных цифр x_s была равна заданному числу (12). Значения n принадлежат специфичному диапазону и априори равновероятны. Однако полученные апостериорные вероятности крайне неоднородны: $P(n=2|sum=12)\approx 0.9$; $P(n=3|sum=12)\approx 0.09$; $P(n=4|sum=12)\approx 0.009$.

Лежащая в основе комбинаторика достаточно очевидна. Однако это тот самый эффект “штрафования сложных решений”, работающий в менее очевидных случаях, например, полиномиальная аппроксимация, использующая полиномы случайной степени или кластеризация с неизвестным числом кластеров.

3.2.2. Принцип минимальной длины описания

Модели универсальной индукции и предсказания основаны на алгоритмической сложности и вероятности [19, 20], которые являются

невычислимыми и не могут быть напрямую применены на практике. Вместо этого на практике обычно применяется принцип минимальной длины описания (сообщения) [21–23]. Вначале эти принципы были введены в некоторой специфичной строгой форме, но сейчас применяются во многих практических методах [24] в форме следующего свободного общего определения: лучшая модель данного источника данных та, которая минимизирует сумму

- длины в битах описания модели;
- длины в битах информации, закодированной с использованием модели.

Главное назначение этих принципов – избежать переобучения путем штрафования моделей на основе их сложности, которая вычисляется с помощью эвристично определенных схем кодирования. Такой “прикладной принцип МДО” довольно полезен, но по большей части в контексте слабого ИИ. Преодоление разрыва между колмогоровской сложностью и приложениями принципа МДО также может быть шагом на пути к преодолением разрыва между слабым и сильным ИИ.

3.2.3. Автоматическая оценка сложности генеративных моделей

Очевидно, если мы хотим, чтобы оптимизационные запросы работали таким же образом, как и семплирующие запросы, мы должны учесть вероятности, с которыми генерируются кандидаты на решения. Здесь мы предполагаем, что критерий, переданный оптимизационным запросам, может быть расценен как минус логарифм правдоподобия. Тогда будет достаточно автоматически вычислять и добавлять минус логарифм априорной вероятности кандидата на решение, чтобы достигнуть желаемого поведения.

Мы вычисляем эти априорные вероятности путем перемножения вероятностей в тех узлах поддерева следов выполнения программы, начиная с *AnnealingQuery* и л и *GPQuery*, в которых делаются базовые случайные выборы. Здесь мы предполагаем, что список выражений, который подается

запросам, релевантный. В результате, каждый такой выбор принимается в расчет только один раз, даже если переменная, ссылаемая на этот выбор, используется много раз.

AnnealingQuery и *GPQuery* были модифицированы и протестированы на программе, представленной выше, и возвращали n=1 во всех случаях, то есть, их поведение соответствует желаемому. Например, в случае трех центров '(-7., -2., 10.) *AnnealingQuery* вернет случайным образом n=1 или n=2, в то время как *GPQuery* вернет их вероятности. Однако оптимизационные запросы могут быть намного более эффективными и могут быть использованы для нахождения отправной точки для таких методов как *mh-query.*

ВЫВОДЫ

Была разработана эффективная реализация интерпретатора вероятностного языка с оптимизационными запросами на опорном языке C++, которая может быть применена к задачам реального мира. Было обнаружено, что оптимизационные запросы, ищущие экстремум заданной целевой функции и при этом игнорирующие априорные вероятности, задаваемые вероятностной программой, могут приводить к известному эффекту «переобучения». Хотя такое поведение является корректным с точки зрения семантики оптимизационных запросов, оно является нежелательным, так как для аналогичных программ семплирующие запросы, строящие апостериорные распределения вероятностей с учетом априорного распределения, к эффекту «переобучения» не приводят.

В разработанный интерпретатор была добавлена возможность автоматической модификации задаваемого в программе критерия в соответствии с принципом минимальной длины описания (то есть с добавлением минус логарифма априорной вероятности решения), что позволило устранить нежелательное поведение оптимизационных запросов.

ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНАЯ ПРОВЕРКА

4.1. Исследование эффективности реализации оптимизационных запросов

Рассмотрим три задачи, каждая из которых может быть поставлена как для условного семплирования, так и для оптимизации фитнес-функции, и сравним три функции запросов – mh-query (web-church), annealing-query и evolution-query (реализация на Scheme). mh-query будет использован для обнаружения только одного экземпляра решения (так как интересно узнать об эффективности на этом шаге; к тому же, в данных задачах не нужны апостериорные распределения, а требуется нахождение единственного решения).

4.1.1. Полиномиальная аппроксимация

Рассмотрим следующую генеративную полиномиальную модель со списком ws неизвестного числа wn параметров с неизвестными значениями, которые должны быть оптимизированы для соответствия наблюдениям.

```
; пример наблюдений
(define xs '(0      1      2        3        4))
(define ys '(0.05 4.9 14.06 29.99  43.97))
; полином(x|ws)
(define (calc-poly x ws)
        (if (null? ws) 0
            (+ (car ws) (* x (calc-poly x (cdr ws))))))
; отобразить xs с помощью calc-poly
(define (generate xs ws)
        (if (null? xs) '()
            (cons (calc-poly (car xs) ws)
                  (generate (cdr xs) ws))))
; сделать список случайных ws
```

```
; (неизвестные параметры полинома)
(define (make-ws n sigma)
          (if (= n 0) '()
             (cons (gaussian 0 sigma)
                   (make-ws (- n 1) sigma))))
; случайное число параметров
(define wn (+ (random-integer (length xs)) 1))
; параметры, возвращаемые запросом
(define ws (make-ws wn 10))
; ys сгенерированные экземпляром модели
(define ys-gen (generate xs ws))
ws
```

Полная генеративная модель также должна включать в себя шум, но такая модель будет практически бесполезной, так как будет невозможно вслепую породить правильные отсчеты шумов. Вместо этого используется среднеквадратичное отклонение (СКО) в annealing-query и evolution-query, и следующее условие используется в mh-query:

```
(define (noisy-equals? x y)
  (flip (exp (* -30 (expt (- x y) 2)))))
(all (map noisy-equals? ys-gen ys))
```

noisy-equals? может случайно быть true, даже если его аргументы различны, но с уменьшающейся вероятностью. Скорость этого уменьшения задается значением, которое равно 30 в примере кода. Чем меньше это значение, тем менее точно выполняется равенство. Мы выбрали такое значение, чтобы время работы mh-query было примерно равно времени работы annealing-query и evolution-query (с контролем времени выполнения путем указания количества итераций), так чтобы можно было сравнить точность решений, найденных разными методами. Конечно, такое сравнение немного нестрогое, но оно качественно адекватно в виду того, что

возрастание времени вычисления приведет только к логарифмическому увеличению точности.

Результаты нескольких функций и информационных точек представлены в табл. 1.

Таблица 1. Среднее СКО

Задача	**СКО**		
	mh-query	***annealing-query***	***evolution-query***
$4x^2+3x$ *xs*=(0 1 2 3)	1.71	0.217	0.035
$4x^2+3x$ *xs*=(0 0.1 0.2 0.3 0.4 0.5)	0.94	0.425	0.010
xs=(0 0.1 0.2 0.3 0.4 0.5)	0.467	0.169	0.007

Как можно увидеть, mh-query здесь неэффективен – ему требуется очень нестрогий noise-equals?, приводящий к неточным результатам. Если сделать это условие строже, происходит огромное увеличение времени вычисления. Evolution-query более точен, в то время как annealing-query работает правильно, но сходится медленнее. Худшая точность достигается, когда неправильно выбирается wn.

Важно увидеть, как работает кроссовер в следах выполнения в “фенотипах” потомков. Рассмотрим пример того, как кроссовер влияет на значения ws

'(1.903864 -11.119573 4.562440) +

'(-20.396958 -12.492696 -0.735389 3.308482) →

'(-5.232313 -11.462677 2.3152821 3.308482)

Значения на одинаковых позициях усреднены со случайными весами (хотя эти веса независимы для различных позиций как в геометрическом семантическом кроссовере). Если длина (то есть wn) родительского вектора параметров ws различна, значения wn потомков будут соотноситься с одним из родительских или будет находиться между ними. Эти результаты кажутся разумными.

4.1.2. Задача суммы подмножеств

В задаче суммы подмножеств дается множество целых чисел, и должно быть найдено непустое подмножество, такое что сумма его элементов равна заданному целочисленному значению (если это значение не равно нулю, то нет необходимости проверять решение на нетривиальность, так что мы предположим, что сумма равна 1 и пропустим эту проверку для простоты). Целые числа в каждом множестве порождены как случайные значения в определенном диапазоне, например, от -10000 до 10000. Выбирается случайное подмножество и последнее число вычисляется как 1 минус сумма элементов этого подмножества. Следующая программа задает генеративную модель для этой задачи.

```
(define xs '(9568 5716 8382 7900 -5461 5087 1138 -1111 -9695 -5468 6345
-1473 -7521 -4323 9893 -9032 -4715 3699 5104 1551))
(define (make-ws n)
    (if (= n 0) '()
        (cons (flip) (make-ws (- n 1)))))
(define ws (make-ws (length xs)))
(define (summ xs ws)
    (if (null? xs) 0
        (+ (if (car ws) (car xs) 0) (summ (cdr xs) (cdr ws)))))
(define subset-sum (summ xs ws))
ws
```

mh-query был выбран, используя условие (equal? subset-sum 1), тогда как annealing-query и evolution-query выполнялись, чтобы минимизировать (abs (- subset-sum 1)). Прямое сравнение различных запросов оказалось сложным в этой задаче. Однако результаты качественно похожи. Все методы либо стабильно находят решения (это тот случай, когда размерность задачи около 15 или меньше, или когда разброс чисел невелик и много подмножеств может давать в сумме желаемое значение), или все методы не преуспевают

(чтобы достигнуть этого, должны быть взяты числа из большего диапазона или взято множество размером 20 или больше).

Однако для задач определенной сложности могут быть получены промежуточные результаты. В частности, следующие результаты были получены для диапазона [-10000, 10000] и размеров выборки 20÷25. mh-query был способен найти 83% верных ответов, в то время, как ему было дано в 2 раза больше времени, чем annealing-query and evolution-query. annealing-query и evolution-query показали примерно 75% верных решений (их производительность варьируется в зависимости от настроек, и annealing-query приводит к более стабильным результатам, в то время как наша простая форма генетического программирования имеет некоторую тенденцию застревать в локальных экстремумах). Стоит также отметить, что в остальных 25% случаев найденное решение почти оптимально (ошибка равна 1).

Убедимся, что эти немногим более слабые результаты генетического программирования не являются следствием некорректного функционирования. Рассмотрим следующий типичный эффект кроссовера над следами выполнения программ на уровне “фенотипа”:

'(#t #f #t #t #t #t #f #t #t) +

'(#f #f #t #t #f #t #f #t #t #f) →

'(#t #f #t #t #t #t #f #f #t #f)

Как можно видеть, это однородное скрещивание. Возможно, это не самый интересный вариант скрещивания, но он работает корректно.

Из этого примера также можно видеть, что оптимизационные запросы в вероятностном программировании подходят для решения детерминистических задач, а не только задач вероятностного вывода.

4.1.3. Предсказание целочисленных последовательностей

Еще одним рассмотренным нами тестовым заданием была задача предсказания целочисленных последовательностей. Мы ограничили

множество возможных последовательностей полиномами, но оно может быть легко расширено в более широкий класс последовательностей, определенных рекуррентными соотношениями. Рассмотрим следующий фрагмент генеративной модели.

```
; выражения генерируются рекурсивно
(define xs '(1 2 3 4 5 6))
(define ys '(3 7 13 21 31 43))
(define (gen-expr)
  (if (flip 0.6)
      (if (flip) 'x (random-integer 10))
      (list (multinomial '(+ - *) '(1 1 1))
            (gen-expr) (gen-expr))))
(define (f x) (eval (list 'let (list (list 'x x)) expr)))
```

После этих определений используется функция f(x) для отображения всех xs и проверки, соответствует ли результат ys или вычисления полного отклонения, в зависимости от типа запроса.

Мы провели тесты для различных последовательностей и сравнили результаты. mh-query не был способен найти решение в каждом запуске. В зависимости от браузера, он заканчивал либо с ошибкой “Превышен максимальный размер стека” или работал очень долго в некоторых запусках. annealing-query и evolution-query также не могли найти точные решения в каждом случае и заканчивали с неточными решениями. Процент запусков, в которых были получены корректные решения, показан в табл. 2. Значения xs были равны '(0 1 2 3 4 5).

mh-query показал здесь удивительно плохие результаты, хотя его вывод над другими рекурсивными моделями может быть успешным. evolution-query также показал результаты немного хуже, чем annealing-query. Причина, вероятно, состоит в том, что задача не раскладывается хорошо на подзадачи, так что кроссовер не приносит преимущества, а имитация отжига может постепенно приближаться к лучшему решению.

Таблица 2. Процент корректных решений

ys	Корректные ответы, %		
	mh-query	***annealing-query***	***evolution-query***
'(0 1 2 3 4 5)	90%	100%	100%
'(0 1 4 9 16 25)	20%	100%	100%
'(1 2 5 10 17 26)	10%	70%	80%
'(1 4 9 16 25 36)	0%	90%	80%
'(1 3 11 31 69 131)	0%	90%	60%

Тем не менее, кажется, что оператор кроссовера над следами выполнения программ производит более разумные результаты в пространстве фенотипов. Если структура родителей соотносится, каждый лист случайно берется от одного из родителей, например, '(+ (+ 3 x) x) + '(- (- x x) x) → '(- (+ x x) x). В узлах, где структура расходится, случайно берется поддерево от одного из родителей, например,

'(- (- (* (* 3 (* x x)) 3) (- x 8)) (* (- x 0) x)) +'(- 3 (- 5 x)) → '(- 3 (* (- x 0) x))

'(* (+ 4 x) x) + '(* (* 2 (- 1 x)) 7) → '(* (* 4 x) 7)

Эффект “фенотипического” кроссовера представляется чем-то нестрогим, но не бессмысленным, и он производит корректные кандидаты на решение, которые наследуют информацию от своих родителей.

4.2. Проверка применимости оптимизационных запросов в задачах с реальными данными

Так как мы ставили целью разработку вероятностного программирования для Тьюринг-полных языков, мы рассматриваем задачи анализа изображений, которые довольно тяжелы для вычисления. На момент реализации нашей библиотеки единственный пример подобного приложения – это работа [18] (и к сожалению, в ней отсутствует информация о времени вычисления). Поэтому возможность решить задачи анализа изображений в разумное время может быть использована как внушительная демонстрация эффективности оптимизационного подхода.

Рассмотрим задачу обнаружения эритроцитов (целью нашей системы не было решение этой конкретной задачи, и она взята просто как пример;

могут быть взяты другие задачи). Типичное изображение представлено на рис. 4. Задача состоит в том, чтобы обнаружить и сосчитать клетки на изображениях мазков крови. Обычно эта задача решается путем детектирования краевых пикселей и применения преобразования Хафа или путем прослеживания контуров и вписывания окружностей [25–28]. Прямого применения существующих реализаций методов обработки изображений недостаточно; необходимо применение нетривиальных комбинаций различных обрабатывающих функций или даже *ad hoc* реализаций этих функций.

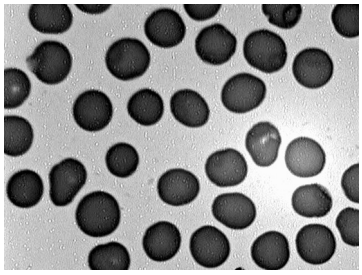

Рис. 4. Оригинальное изображение с красными кровяными клетками

Однако приемлемое решение может быть получено, используя следующую очень маленькую генеративную модель:

```
Define(n, RndInt(20) + 10)
Define(circs, Repeat(n, Lambda0(List(RndInt(img.cols),
        RndInt(img.rows),
        RndInt(12)+6))))
Define(gen, Foldr(Lambda(circ, im,
      DrawCircle(im, circ[0], circ[1], circ[2], V(168), V(-1))),
      circs, V(cv::Mat::zeros(img.rows, img.cols, cv::CV_8UC1))))
circs
Log(MatDiff2(gen, V(img))) * V(img.cols * img.rows)
```

Здесь *n* – число окружностей, которые надо нарисовать, *circs* – список случайных центров окружностей и радиусов (*i m g* – инвертированное изображение для анализа), *g e n* – сгенерированное изображение. Оно генерируется, начиная с пустого изображения, последовательно рисуя окружности из *circs*. Следует отметить, что так как наша библиотека

реализует функциональный квази-язык, такие функции как *DrawCircle* не изменяют данное изображение, а возвращают новое. Последние два выражения в модели содержат результирующее значение и оценку минус логарифма правдоподобия. Чтобы улучшить качество работы, мы также реализовали класс *Drawer*. Во время оценки *Drawer* обрабатывает список фигур и рисует их, используя одно выходное изображение. Была протестирована программа с *Drawer* вместо *Foldr* and *DrawCircle.*

AnnealingQuery не справляется на картинках с большим числом объектов, так как каждый шаг имитации отжига состоит в попытке модифицировать координаты и размеры всех окружностей одновременно, и успешная модификация становится маловероятной для большого числа переменных. *GPQuery* показал приемлемые результаты (см. рис. 5), но с некоторым уточнением оператора кроссовера.

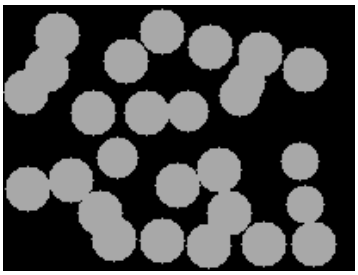

Рис. 5. Результат работы *GPQuery* (размер популяции = 300, число поколений = 100, величина мутаций = 0,005)

GPQuery ведет к лучшим результатам здесь, так как метод автоматически производит “мягкую декомпозицию” данной задачи. Однако его результаты не оптимальны, и время поиска не так чтобы невелико (5-30 секунд на i5 2.6 GHz в зависимости от параметров генетического программирования). Тем не менее, он уже может использоваться для быстрого прототипирования.

Проблема поиска – одна из наиболее важных здесь, и она далека от того, чтобы быть полностью решенной. Однако мы заинтересованы в тестировании разработанного метода для включения критерия МДО в оптимизационные запросы. Рассмотрим вычисленные значения этого критерия на различных небольших изображениях (рис. 6) для разного числа

окружностей с целью гарантировать, что найденное решение близко к оптимальному. Табл. 3 подытоживает полученные результаты.

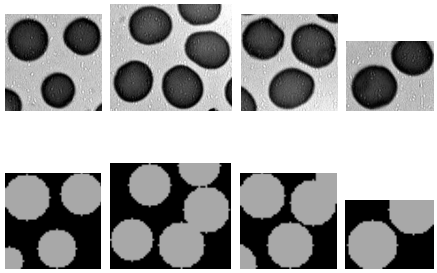

Рис. 6. Фрагменты изображений и результатов над ними

Таблица 3. Полная длина описания, биты

Изображение #	*n*					
	1	2	3	4	5	6
1	14650.4	14038.0	13131.2	12687.3	12689.3	12690.0
2	20201.3	19612.1	18888.2	17955.2	17104.2	17115.2
3	14680.3	13995.2	12808.1	12391.7	12316.6	12321.0
4	9270.7	8155.1	8160.6	8162.6	8163.2	8168.5

Здесь можно видеть, что общая длина описания начинает медленно возрастать от некоторого числа окружностей для каждого изображения. Каждая окружность добавляет около 10 бит длины описания. Так что минус логарифм правдоподобия немного убывает, но медленнее, чем возрастает сложность. В самом деле, в виду того, что кровяные клетки не идеально круглые, дополнительные окружности, вписанные в непокрытые части клеток, могут увеличивать сложность модели меньше, чем уменьшать минус логарифм правдоподобия в некоторых случаях. Однако в этих случаях запросы, вычисляющие апостериорную вероятность, также будут давать выраженный пик на том же числе окружностей. Другими словами, причина этого результата не в процедурах запросов или критерии, а в модели. В общем случае, найденный минимум критерия длины описания соответствует реальному числу кровяных клеток и частично представленные клетки надежно детектируются.

ВЫВОДЫ

Была проведена экспериментальная проверка разработанных оптимизационных запросов на задачах полиномиальной аппроксимации, сумме подмножеств, предсказании целочисленных последовательностей, анализе изображений мазков крови.

Проверка подтвердила корректность реализации оптимизационных запросов, включая корректность работы метаэвристических методов поиска и автоматического вычисления критерия минимальной длины описания.

Оптимизационные запросы в целом показали более высокую (для некоторых задач – существенно более высокую) эффективность по сравнению с семплирующими запросами.

ЗАКЛЮЧЕНИЕ

Мы разработали методы имитации отжига и генетического программирования над следами выполнения программ. Насколько нам известно, это первая реализация этих методов таким образом.

Были использованы одинаковые функции для генетических операторов над следами выполнения программ для решения оптимизационных задач очень разных типов объектов, включая параметрически определенные функции, множества, символьные выражения и даже изображения без производства некорректных кандидатов на решения. Наша реализация соответствует однородному кроссоверу. Должны быть реализованы другие типы генетических операторов, так как наша реализация показывает преимущество над имитацией отжига только в задачах, где выучиваются модели с вещественными значениями. Интересно совместить вероятностное программирование и такие продвинутые системы генетического программирования, как MOSES [15].

Несмотря на простоту использованных метаэвристических методов поиска, они показали лучшие результаты, чем стандартный mh-query. Хотя это сравнение не означает, что annealing-query или evolution-query могут заменить mh-query, так как они решают несколько разные задачи, но показывает, что они могут быть совмещены, и также оптимизационные зарпосы могут быть полезны для расширения семантики вероятностных языков программирования. До сих пор остается недостаточной эффективность общих методов вывода, и это может быть одним из принципиальных препятствий на пути к сильному ИИ. Возможно, один общий метод вывода не может быть эффективен во всех предметных областях, так что метод вывода должен быть автоматически специализирован по отношению к каждой области, встреченной агентом сильного ИИ [29, 30], которая предполагает, что такие методы должны быть тесно связаны с когнитивными архитектурами.

Разработанный метод автоматического использования принципа минимальной длины описания в вероятностном программировании как сокращает разрыв между свободно применимым принципом МДО и основанным на теории и непрактичной колмогоровской сложности, так и помогает избежать переобучения в оптимизационных запросах, делая их эффективной альтернативой более традиционным запросам, оценивающим условные вероятности. Были проведены эксперименты на примере задачи анализа изображений, подтверждающие приемлемость этого подхода.

Однако даже неспециализированные оптимизационные запросы не смогли эффективно решить произвольные задачи индукции, особенно связанные с общим ИИ. В самом деле, задача такого эффективного вывода может сама по себе быть рассмотрена как ИИ-полная задача. Поэтому должны быть установлены более глубокие связи между общим ИИ и областью вероятностного программирования.

СПИСОК ЛИТЕРАТУРЫ

1. Goodman N.D., Tenenbaum J.B. Probabilistic models of cognition. [Электронный ресурс]. Режим доступа: http://probmods.org, свободный. Яз. англ. (дата обращения 15.03.2016).
2. Luke S. Essentials of metaheuristics / Lulu. 2009. 235 p.
3. Newell A., Shaw J.C., Siman H. Empirical explorations of the Logic Theory Machine: a case study in heuristics // proc. Western Joint Computer Conference. 1957. P. 218-239.
4. Brachman R. Getting Back to “The Very Idea” // AI Magazine. 2005. V. 26. No 4. P. 48–50.
5. Nilsson N.J. Human-Level Artificial Intelligence? Be Serious! // AI Magazine. 2005. V. 26. No 4. P. 68–75.
6. Duch W., Oentaryo R.J., Pasquier M. Cognitive Architectures: Where Do We Go from Here // Frontiers in Artificial Intelligence and Applications (Proc. 1st AGI Conference). 2008. V. 171. P. 122–136.
7. Hutter M. Universal Artificial Intelligence. Sequential Decisions Based on Algorithmic Probability / Springer. 2005. 278 p.
8. Minka, T., Winn, J.M., Guiver, J.P., Knowles, D.: Infer.NET 2.4. Microsoft Research Camb. [Электронный ресурс]. Режим доступа: http://research.microsoft.com/infernet, свободный. Яз. англ. (дата обращения 15.03.2016).
9. Koller D., McAllester D.A., Pfeffer A. Effective Bayesian inference for stochastic programs // Proc. National Conference on Artificial Intelligence (AAAI). 1997. P. 740–747.
10. Stuhlmüller A., Goodman N. D. A dynamic programming algorithm for inference in recursive probabilistic programs // In: Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12). 2012. arXiv:1206.3555 [cs.AI].
11. Goodman N.D., Mansinghka V.K., Roy D.M., Bonawitz K., Tenenbaum J.B. Church: a language for generative models // Proc. of Uncertainty in Artificial Intelligence. 2008. arXiv:1206.3255 [cs.PL].

12. Milch B., Russell S. General-purpose MCMC inference over relational structures // Proc. 22nd Conference on Uncertainty in Artificial Intelligence. 2006. P. 349–358.
13. Perov Y., Wood F. Learning Probabilistic Programs. // arXiv:1407.2646 [cs.AI]. 2014.
14. Solomonoff R. Algorithmic Probability, Heuristic Programming and AGI // In: Baum, E., Hutter, M., Kitzelmann, E. (Eds). Advances in Intelligent Systems Research. 2010. V. 10 (proc. 3rd Conf. on Artificial General Intelligence). P. 151–157.
15. Goertzel B., Geisweiller N., Pennachin C., Ng K. Integrating Feature Selection into Program Learning // In: Kühnberger, K.-W., Rudolph, S., Wang, P. (Eds.): AGI'13, LNAI. 2013. V. 7999. P. 31–39.
16. McDermott J., Paula C. Program Optimisation with Dependency Injection // Proc. 16th European Conference on Genetic Programming, EuroGP. 2013.
17. Gordon A.D., Henzinger Th.A., Nori A.V., Rajamani S.K. Probabilistic programming // International Conference on Software Engineering. 2014.
18. Mansinghka V., Kulkarni T., Perov Y., Tenenbaum J. Approximate Bayesian Image Interpretation using Generative Probabilistic Graphics Programs // Advances in Neural Information Processing Systems. 2013. arXiv:1307.0060 [cs.AI].
19. Hutter M. Universal Algorithmic Intelligence: A Mathematical Top→Down Approach // in Artificial General Intelligence. Cognitive Technologies, B. Goertzel and C. Pennachin (Eds.). Springer. 2007. P. 227–290.
20. Solomonoff R. Does Algorithmic Probability Solve the Problem of Induction? // Oxbridge Research, P.O.B. 391887, Cambridge, Mass. 02139 1997.
21. Wallace C.S., Boulton D.M. An Information Measure for Classification // Computer Journal. 1968. V. 11. P. 185–195.
22. Rissanen J.J. Modeling by the Shortest Data Description // Automatica-J.IFAC. 1978. V. 14. P. 465–471.

23. Vitanyi P.M.B., Li M. Minimum Description Length Induction, Bayesianism, and Kolmogorov complexity // IEEE Trans. on Information Theory. 2000. V. 46 (2). P. 446–464.
24. Potapov A.S. Principle of Representational Minimum Description Length in Image Analysis and Pattern Recognition // Pattern Recognition and Image Analysis. 2012. V. 22 (1). P. 82–91.
25. Zhdanov I.N., Potapov A.S., Shcherbakov O.V. Erythrometry method based on a modified Hough transform // Journal of Optical Technology. 2013. V. 80. No. 3. P. 201–203.
26. Maitra M., Gupta R.K., Mukherjee M. Detection and counting of red blood cells in blood cell images using Hough transform // International journal of computer applications. 2012. V. 53. № 16. P. 18–22.
27. Poomcokrak J., Neatpisarnvanit C. Red blood cells extraction and counting // The 3rd International Symposium on Biomedical Engineering. 2008. P. 199–203.
28. Kimbahune V.V., Ukepp N.J. Blood cell image segmentation and counting // International journal of engeneering science and technology. 2011. V. 3. № 3. P. 2448–2453.
29. Khudobakhshov V. Metacomputations and Program-based Knowledge Representation // K.-U. Kühnberger, S. Rudolph, P. Wang (Eds.): AGI'13. Lecture Notes in Artificial Intelligence. 2013. V. 7999. P. 70–77.
30. Potapov A., Rodionov S. Making Universal Induction Efficient by Specialization // B. Goertzel et al. (Eds.): AGI 2014. Lecture Notes in Artificial Intelligence. 2014. V. 8598. P. 133–142.